Overview of the Scalable Checkpoint / Restart (SCR) Library

Wednesday, October 14, 2009

Adam Moody

moody20@llnl.gov

S&T Principal Directorate - Computation Directorate

This work performed under the auspices of the U.S. Department of Energy by
Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Lawrence Livermore National Laboratory

Background

- Livermore has many applications which run at large scale for long times, so failures are a concern.
- Even on a failure-free machine, running jobs are routinely interrupted at the end of 12 hour time slice windows.
- To deal with failures and time slice windows, applications periodically write out checkpoint files from which they restart (a.k.a. restart dumps).
- Typically, these checkpoints are coordinated, and they are written as a file-per-process or they can be configured to be so.

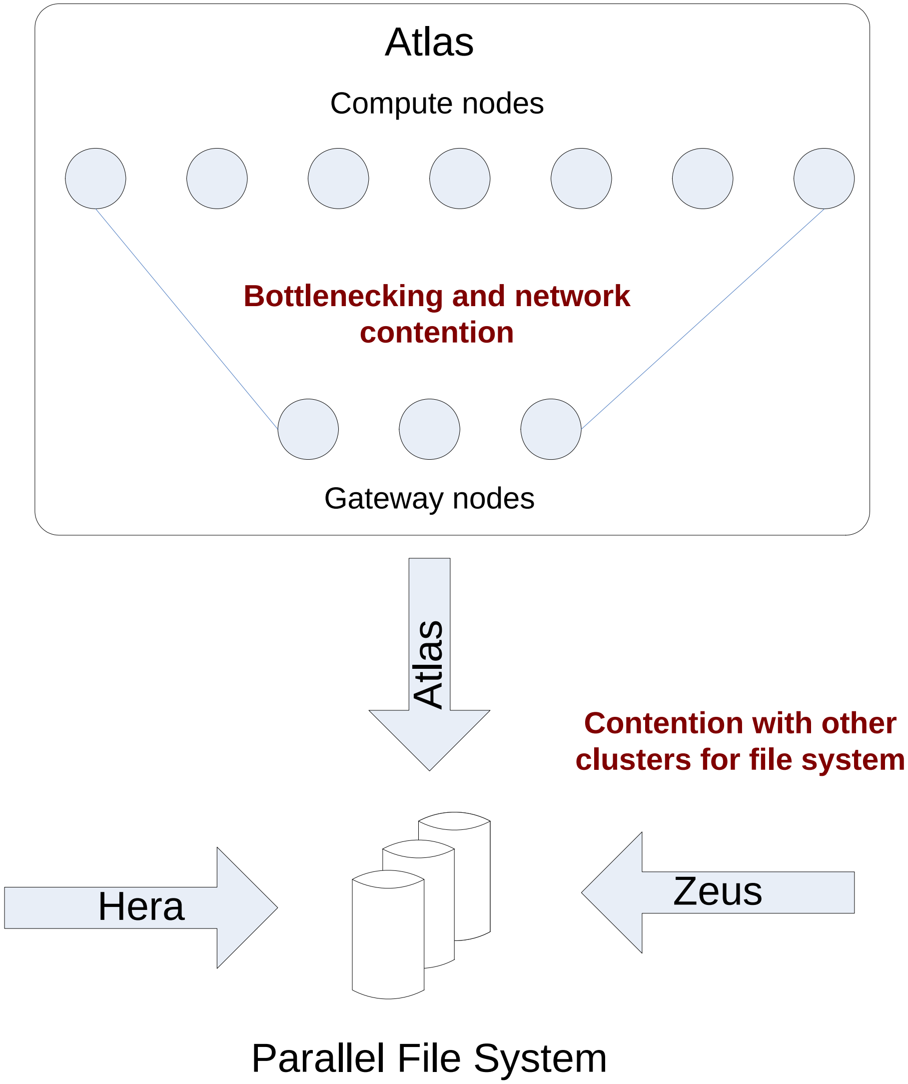
Motivation

- During the early days of Atlas, before certain hardware and software bugs were worked out of the system, it was necessary to checkpoint large jobs frequently to make progress.
- A checkpoint of pf3d on 4096 processes (512 nodes) of Atlas typically took 20 minutes and could be as high as 40 minutes → costly, so configured run to checkpoint every 2 hours.
- However, the mean time before failure was only about 4 hours, and many runs failed before writing a checkpoint → lots of lost time.

Motivation (cont.)

- Observations:
  - Only need the most recent checkpoint data.
  - Typically just a single node failed at a time.
- Idea:
  - Store checkpoint data redundantly on compute cluster; only write it to the parallel file system upon a failure.
- This approach can use the full network and parallelism of the job's compute resources to cache checkpoint data.
  - With 1GB/s links, a 1024-node job has 1024GB/s bandwidth.
  - Compares to ~10-20GB/s from parallel file system.

Avoids two problems



Implementation overview

- Design
  - Cache checkpoint data in files on storage on the compute nodes.
  - Run commands after job to flush latest checkpoint to the parallel file system.
  - Define a simple, portable API to integrate around an application's existing checkpoint code.

- Advantages
  - Perfectly scalable → each compute node adds another storage resource.
  - Files persist beyond application processes, so no need to modify how MPI library deals with process failure.
  - Same file format and file name application currently uses, so little impact to application logic.

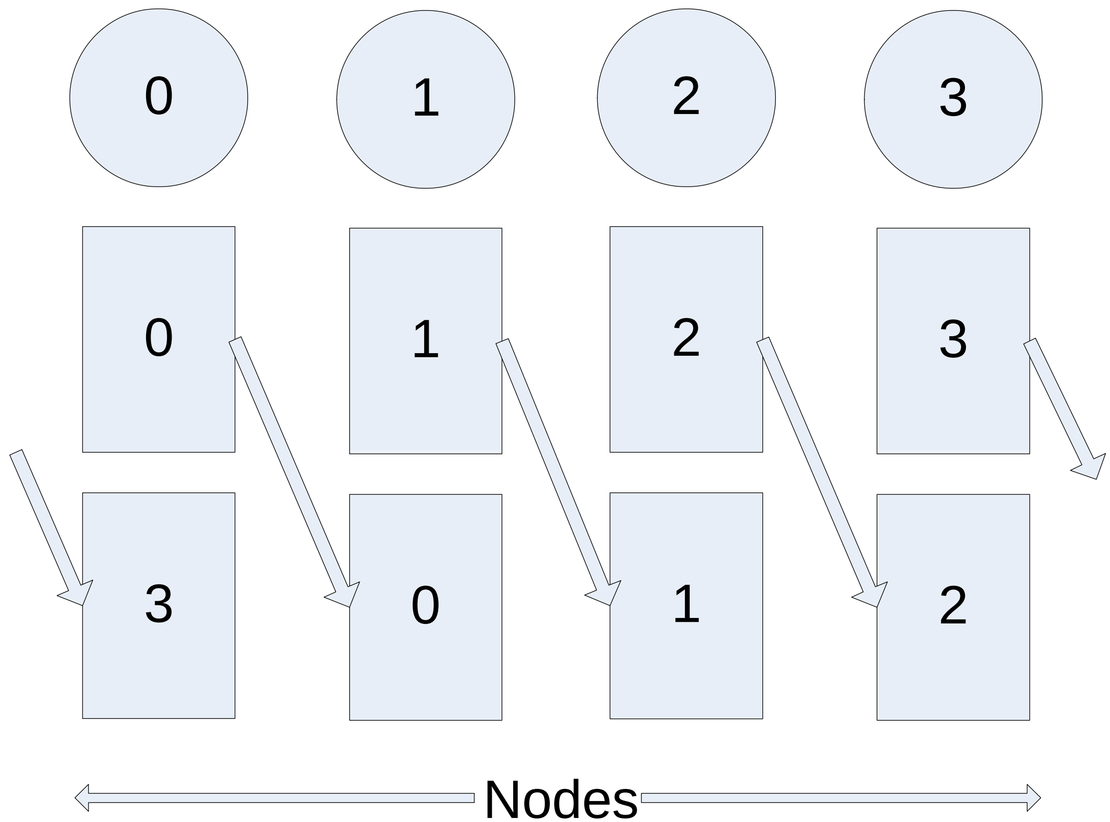
- Disadvantages
  - Only storage available on some systems is RAM disc, for which checkpoint files will consume main memory.
  - Nodes may fail, so need to store files redundantly.
  - Susceptible to catastrophic failure, so need to write to parallel file system occasionally.

Partner-copy redundancy

MPI processes

Write checkpoint file locally on node

Copy checkpoint file to a partner node



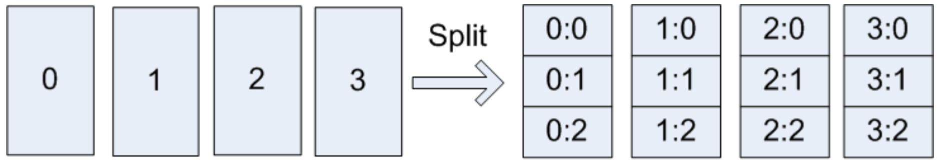
Partner summary

- Can withstand multiple failures, so long as a node and its partner do not fail simultaneously.
- But… it uses a lot of memory.
  - For a checkpoint file of B bytes, requires 2*B storage, which must fit in memory (RAM disc) along with application working set.

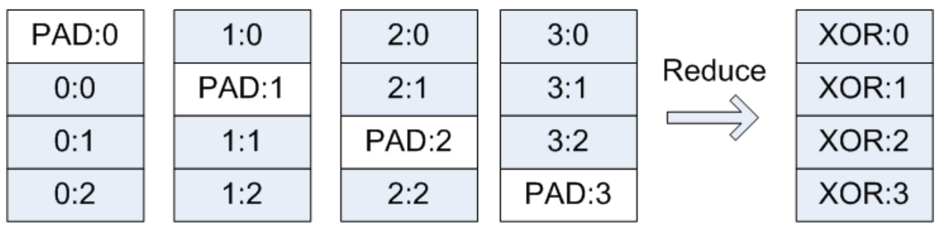
Reducing storage footprint

- Partner worked well and was used during the Atlas DATs starting in late 2007.
- Application working sets required more of main memory by mid-2008.
- Motivated XOR scheme (like RAID-5):
  - Compute XOR file from a set of checkpoints files from different nodes.
  - In a failure, can recover any file in the set using XOR file and remaining N-1 files.
  - Similar to: William Gropp, Robert Ross, and Neill Miller. "*Providing Efficient I/O Redundancy in MPI Environments*", In Lecture Notes in Computer Science, 3241:77–86, September 2004. 11th European PVM/MPI Users' Group.

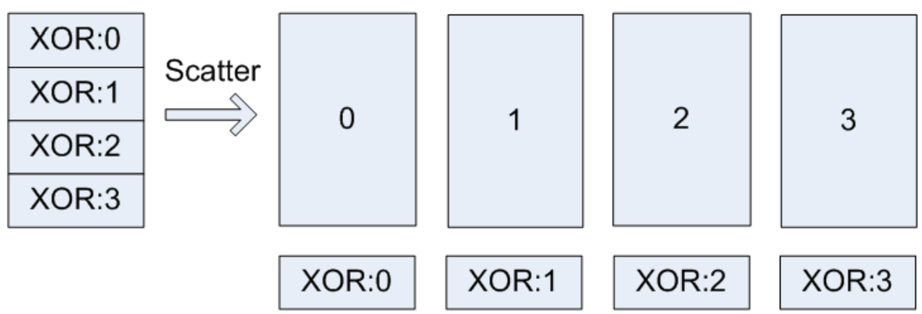
XOR redundancy (similar to RAID5)



Logically split checkpoint files from ranks
on N different nodes into N-1 segments



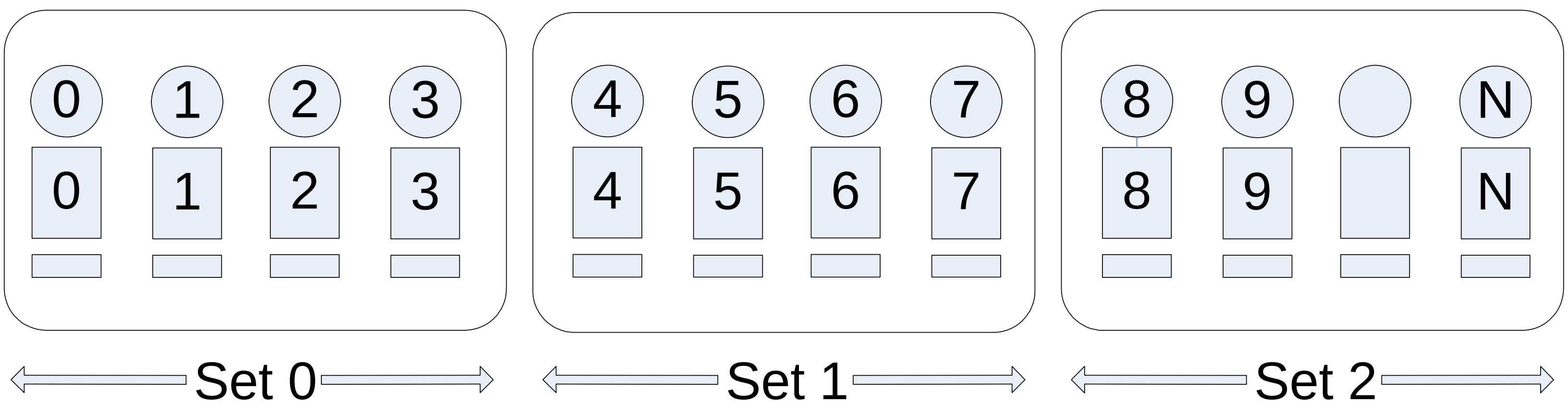
Logically insert alternating zero-padded blocks and reduce



Scatter XOR segments to different nodes

XOR redundancy (cont)

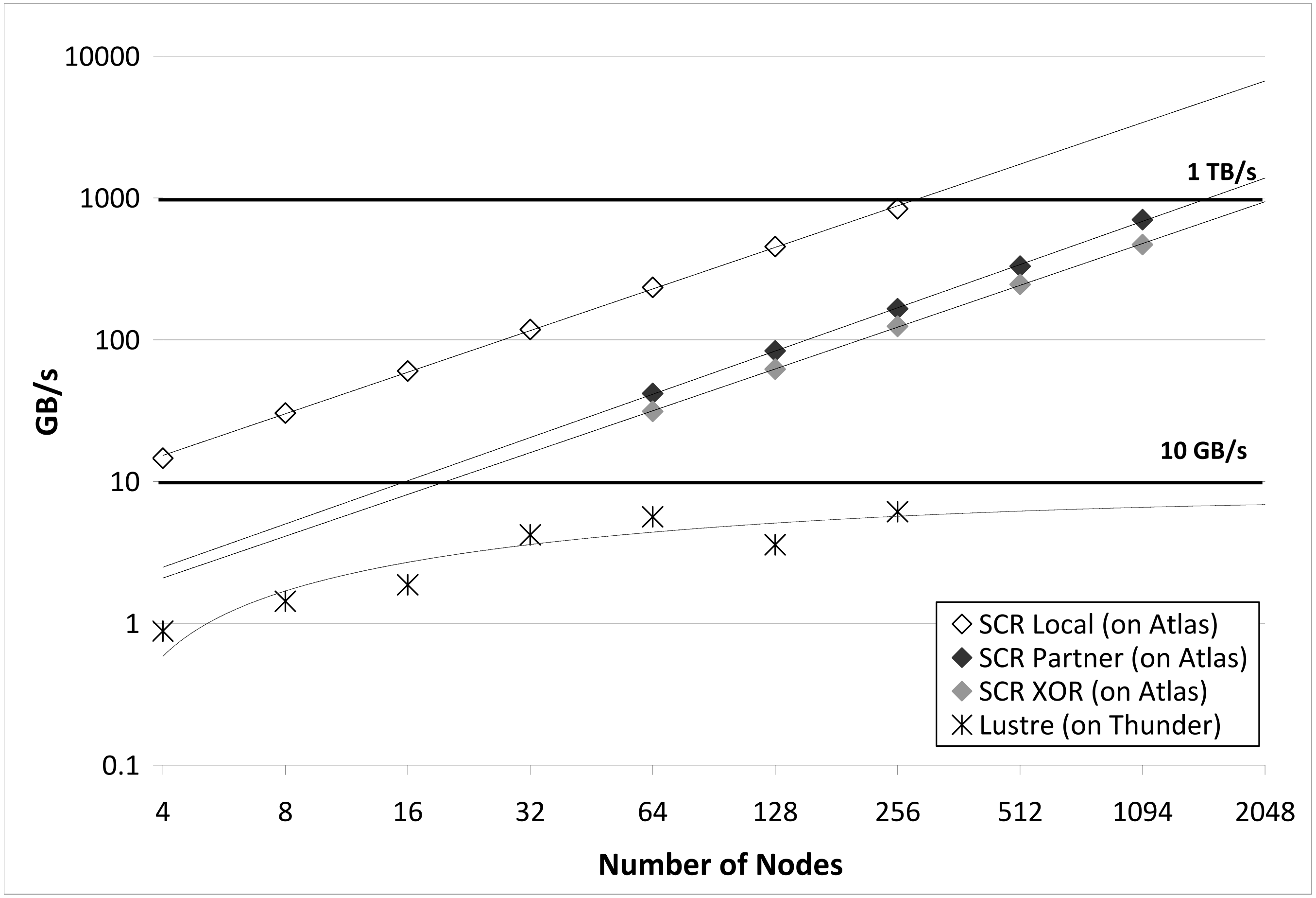
- Break nodes for job into smaller sets, and execute XOR reduce scatter within each set.
- Can withstand multiple failures so long as two nodes in the same set do not fail simultaneously.



XOR summary

- If a checkpoint file is B bytes, requires B+B/(N-1), where N is the size of the XOR set.
  - With Partner, we need 2 full copies of each file.
  - With XOR, we need 1 full copy + some fraction.
- But… it may take longer.
  - Requires more time (or effort) to recover files upon a failure.
  - Slightly slower checkpoint time than Partner on RAM disc (additional computation).
  - XOR can be faster if storage is slow, e.g., hard drives, where drive bandwidth is the bottleneck.

Benchmark checkpoint times



pf3d minimum checkpoint times for 4096 processes

Machine & lscratch	Nodes & Data	Lustre time & BW	SCR time & BW	Speedup
Juno /p/lscratch3	256 nodes 1.88 TB	175 s 10.7 GB/s	13.7 s 140 GB/s	13x
Hera /p/lscratchc	256 nodes 2.07 TB	300 s 7.07 GB/s	15.4 s 138 GB/s	19x
Coastal /p/lscratchb	512 nodes 2.15 TB	392 s 5.62 GB/s	5.80 s 380 GB/s	68x

Scalable restart

- The commands that run after a job to copy the checkpoint files to the parallel file system rebuild lost files after a failure so long as the redundancy scheme holds. This enables one to restart from the parallel file system after a failure.
- However, in many cases, just a single node fails, there is still time left in the job allocation, and all of the checkpoint files are still cached on the cluster.
- Wouldn't it be slick if we could just bring in a spare node, rebuild any missing files, and restart the job in the same allocation without having to write out and read in the files via the parallel file system?

Partner scalable restart

Run job with spare node S.

Job stores checkpoint.

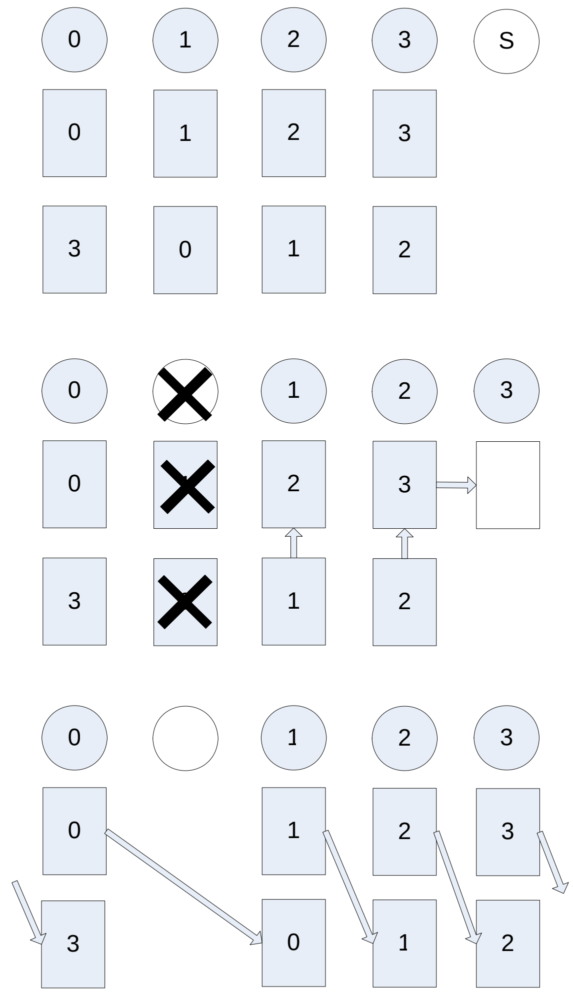
Node dies.

Relaunch with spare.

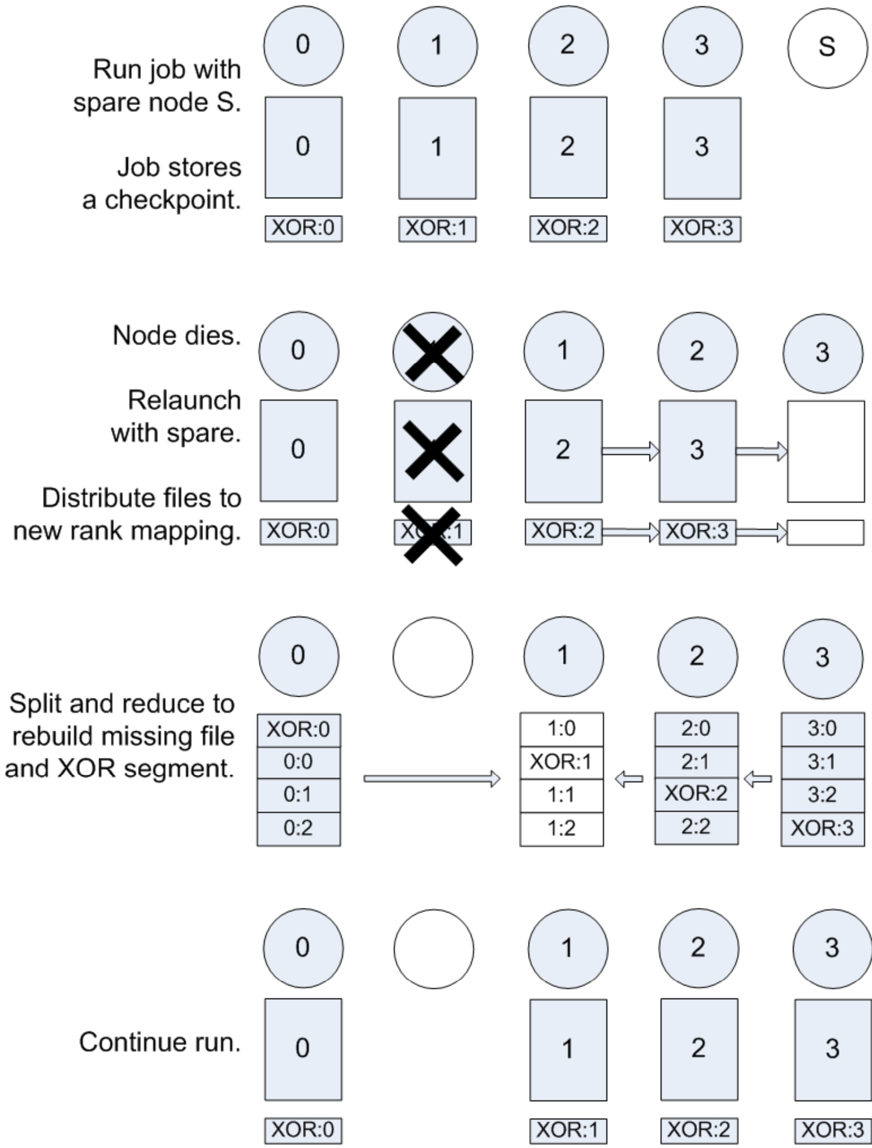
Distribute files to new rank mapping.

Rebuild redundancy.

Continue run.



XOR scalable restart



Value of scalable restart

- Consider three configurations of the same application
  - Without SCR
    - 20 min checkpoint to parallel file system every 200 minutes for an overhead of 10%
  - With SCR (using only scalable checkpoints)
    - 20 sec checkpoint to SCR every 15 minutes for an overhead of 5%
  - With SCR (using scalable checkpoint s and scalable restarts)
    - Checkpoint same as above, 30 sec file rebuild time
- Assume the run hits a node failure half way between checkpoints, and assume it takes the system 5 minutes to detect the failure.
- How long does it take to get back to the same point in the computation in each case?

Value of scalable restart (cont)

	Without SCR	SCR (checkpoint only)	SCR (checkpoint & restart)
Time for system to detect the failure	5 min	5 min	5 min
Time to read checkpoint files during restart	20 min read from parallel file system	20 min write + 20 min read via parallel file system	0.5 min SCR rebuild
Lost compute time that must be made up	100 min	7.5 min	7.5 min
Total time lost	125 min	52.5 min	13 min

The SCR API

```
// Include the SCR header
#include “scr.h”

// Start up SCR (do this just after MPI_Init)
SCR_Init();

// ask SCR whether a checkpoint should be taken
SCR_Need_checkpoint(&flag);

// tell SCR that a new checkpoint is starting
SCR_Start_checkpoint();

// register file as part of checkpoint and / or
// get path to open a checkpoint file
SCR_Route_file(name , file);

// tell SCR that the current checkpoint has completed
SCR_Complete_checkpoint(valid);

// Shut down SCR (do this just before MPI_Finalize)
SCR_Finalize();
```

Using the SCR API: Checkpoint

```
// Determine whether we need to checkpoint
int flag;
SCR_Need_checkpoint(&flag);
if (flag) {
  // Tell SCR that a new checkpoint is starting
  SCR_Start_checkpoint();

  // Define the checkpoint filename for this process
  int rank;
  char name[256];
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  sprintf(name, “rank_%d.ckpt”, rank);

  // Register our file, and get the full path to open it
  char file[SCR_MAX_FILENAME];
  SCR_Route_file(name, file);

  // Open, write, and close the file
  int valid = 0;
  FILE* fs = open(file, “w”);
  if (fs != NULL) {
    valid = 1;
    size_t n = fwrite(checkpoint_data, 1, sizeof(checkpoint_data), fs);
    if (n != sizeof(checkpoint_data)) { valid = 0; }
    if (fclose(fs) != 0) { valid = 0; }
  }

  // Tell SCR whether this process succeeded in writing its checkpoint
  SCR_Complete_checkpoint(valid);
}
```

Case study: pf3d on Juno

- With parallel file system only
  - Checkpoint every 2 time steps at average cost of 1200 secs.
- With parallel file system & SCR
  - Checkpoint every time step at average cost of 15 secs.
  - Write to parallel file system every 14 time steps.
  - Allocate 3 spare nodes for a 256 nodes job
- In a given period
  - 7 times less checkpoint data to parallel file system.
  - Percent time spent checkpointing reduced from 25% to 5.3%.
  - Time lost due to a failure dropped from 55 min to 13 min.
- A nice surprise
  - With SCR, mean time before failure increased from a few hours to tens of hours or even days.
  - In this case, less stress on the network and the parallel file system reduced failure frequency.
  - Far fewer restarts → far less time spent re-computing the same work.

What can SCR do?

- Write checkpoints (up to 100x) faster than the parallel file system.
  - Checkpoint more often → save more work upon failure.
  - Reduce defensive I/O time → increase machine efficiency.
- Reduce load on the parallel file system (community benefit).
  - Each application writing checkpoints to SCR frees up bandwidth to the parallel file system for other jobs.
- Make full use of each time slot via spare nodes.
  - Avoid waiting in the queue for a new time slot after hitting a failed node or process.
- Improve system reliability by shifting checkpoint I/O workload to hardware better suited for the job.

Costs and limitations

- Files are only accessible to the MPI rank which wrote them.
  - Limited support for applications that need process-global access to checkpoint files, which includes applications that can restart with a different number of processes between runs.
- Need hardware and a file system to cache checkpoint files.
  - RAM disc is available on most Linux machines, but at the cost of giving up main memory.
  - Hard drives could be used, but drive reliability is a concern.
- Only 6 functions in the SCR API, but integration may not be trivial.
  - Integration times thus far have ranged from 1 hour to 4 days.
  - Then testing is required.

Ongoing work

- Integrating SCR into more codes at Livermore, and porting SCR to more platforms.
- Automating data collection of performance, failure rates, and file rebuild success rates.
- Using Coastal to investigate effectiveness of solid-state drives for checkpoint storage.
    - 1152-node cluster with a 32GB Intel X-25E SSD mounted on each node.
    - Early testing shows good performance and scalability. Drive performance and reliability over time are open questions.

Interested?

- Open source project:
  - BSD license
  - To be hosted at sourceforge.net/projects/scalablecr

- Should work without much trouble on Linux clusters
  - Depends on a couple other open source packages.

- Email me:
  - Adam Moody
  - moody20@llnl.gov

Extra slides

SCR interpose library

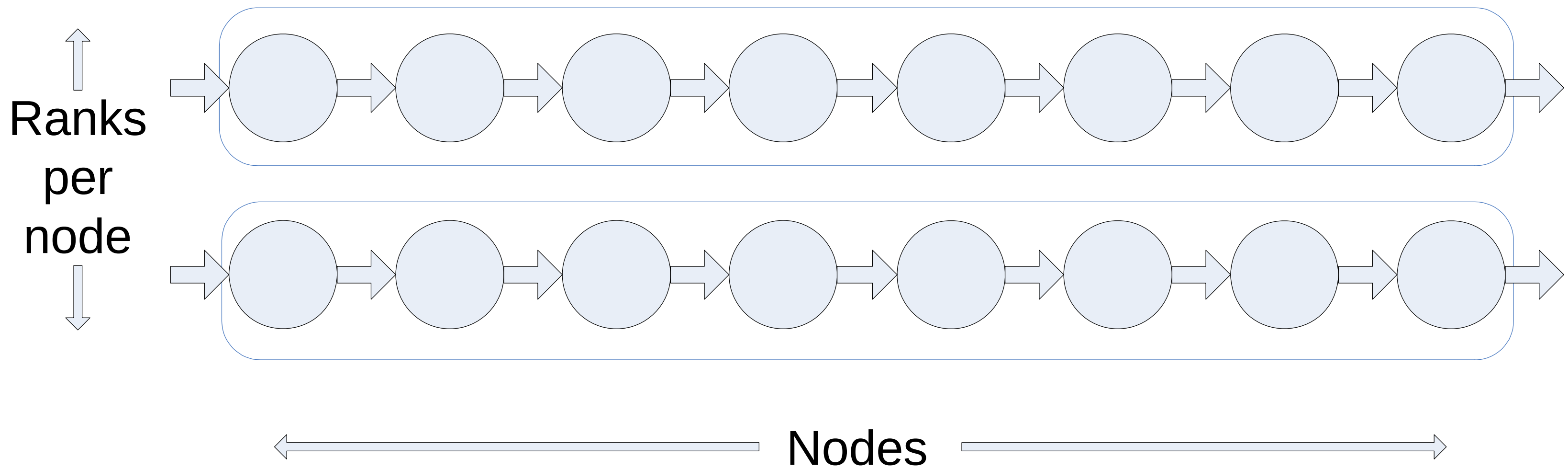
- In some cases, codes can use SCR transparently without even rebuilding.
- For codes that meet certain conditions, one can specify a checkpoint filename via a regular expression and then LD_PRELOAD an SCR interpose library.
- This library intercepts calls to MPI_Init(), open(), close(), and MPI_Finalize(); and then it make calls to SCR library as needed for filenames which match the regular expression.

Catastrophic failures

- Example catastrophic failures from which the library can not recover all checkpoint files
  - Multiple node failure which violates the redundancy scheme (happened twice in the past year).
  - Failure during a checkpoint (~1 in 500 checkpoints).
  - Failure of the node running the job batch script.
  - Parallel file system outage (any Lustre problems).
- To deal with catastrophic failures, it is necessary to write to Lustre occasionally, but much less frequently than without SCR

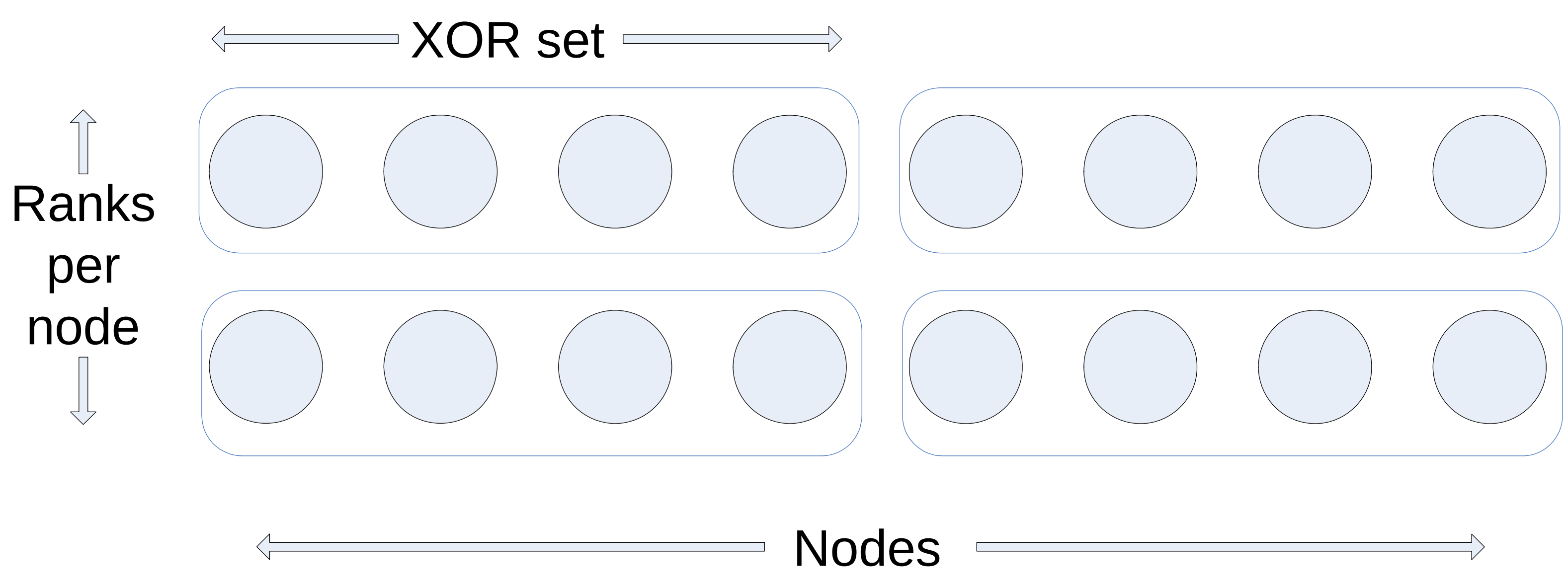
Partner selection

Picking partner ranks
in an 8 node job with 2 ranks per node



XOR set selection

Assigning ranks to XOR sets of size 4
in an 8 node job with 2 ranks per node



Example job script w/ SCR

```
#!/bin/bash
#MSUB –l partition=atlas
#MSUB –l nodes=66
#MSUB –l resfailpolicy=ignore
# above, tell MOAB / SLURM to not kill job allocation upon a node failure
# also note that the job requested 2 spares – it uses 64 nodes but allocated 66

# add the scr commands to the job environment
. /usr/local/tools/dotkit/init.sh
use scr

# specify where checkpoint directories should be written
export SCR_PREFIX=/p/lscratchb/username/run1/checkpoints

# instruct SCR to flush to the file system every 20 checkpoints
export SCR_FLUSH=20

# exit if there is less than hour remaining (3600 seconds)
export SCR_HALT_SECONDS=3600

# attempt to restart the job up to 3 times
export SCR_RETRIES=3

# run the job with scr_srun
scr_srun –n512 –N64 ./my_job
```

Halting an SCR job

- It is important to not stop an SCR job if it is in the middle of a checkpoint, since in this case, there is no complete checkpoint set
- $SCR_HALT_SECONDS + libyogrt can be used to halt the job after a checkpoint before the allocation time expires
- scr_halt command writes a file which the library checks for after each checkpoint
  - ```
    scr_halt
    [--checkpoints <num>]
    [--before <time>]
    [--after <time>]
    [--immediate]
    [--seconds <num>]
    <jobid>
    ```
    ```